Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Learning Framework for Automatic Assume-Guarantee Verification *

Jamieson M. Cobleigh[1], **Dimitra Giannakopoulou**[2], **Corina S. Păsăreanu**[3]

[1] Department of Computer Science, University of Massachusetts, Amherst, MA 01003-9264, USA

e-mail: `jcobleig@cs.umass.edu`

[2] RIACS/USRA, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA

e-mail: `dimitra@email.arc.nasa.gov`

[3] Kestrel Technology LLC, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA

e-mail: `pcorina@email.arc.nasa.gov`

Received: date / Revised version: date

Abstract. Compositional verification is a promising approach to addressing the state explosion problem associated with model checking. One compositional technique advocates proving properties of a system by checking properties of its components in an assume-guarantee style. However, the application of this technique is difficult because it involves non-trivial human input. This paper presents a novel framework for performing assume-guarantee reasoning in an incremental and fully automated fashion. To check a component against a property, our approach generates assumptions that the environment needs to satisfy for the property to hold. These assumptions are then discharged on the rest of the system. Assumptions are computed by a learning algorithm. They are initially approximate, but become gradually more precise by means of counterexamples obtained by model checking the component and its environment, alternately. This iterative process may at any stage conclude that the property is either true or false in the system. We have implemented our approach in the LTSA tool and discuss its application to a NASA system.

1 Introduction

Our work is motivated by an ongoing project at NASA Ames Research Center on the application of model checking to the verification of autonomous software. Autonomous software involves complex concurrent behav-

* This paper is an extended and revised version of [10].

iors for reacting to external stimuli without human intervention. Extensive verification is a pre-requisite for the deployment of missions that involve autonomy.

Given some formal description of a system and of a required property, model checking automatically determines whether the property is satisfied by the system. The limitation of the approach, referred to as the "state explosion" problem, is the exponential relation of the number of states in the system under analysis to the number of components of which the state is made [28]. Model checking therefore does not scale, in general, to systems of realistic size.

Compositional verification presents a promising way of addressing state explosion. It advocates a "divide and conquer" approach where properties of the system are decomposed into properties of its components, so that if each component satisfies its respective property, then so does the entire system. Components are therefore model checked separately. It is often the case, however, that components only satisfy properties in specific contexts (also called environments). This has given rise to the assume-guarantee style of reasoning [25, 30].

Assume-guarantee reasoning first checks whether a component M guarantees a property P, when it is part of a system that satisfies an assumption A. Intuitively, A characterizes all contexts in which the component is expected to operate correctly. To complete the proof, it must also be shown that the remaining components in the system, i.e., M's environment, satisfy A. Several frameworks have been proposed [9, 20, 22, 25, 30, 33] to support this style of reasoning. However, their practical impact has been limited because they require non-trivial human input in defining assumptions that are strong enough to eliminate false violations, but that also reflect the remaining system appropriately.

In contrast, this paper presents a novel framework for performing assume-guarantee reasoning in an *incremental* and *fully automatic* fashion. Our approach iterates a process based on gradually *learning* assumptions. The learning process is based on queries to component M and on counterexamples obtained by model checking M and its environment, alternately. Each iteration may conclude that the required property is satisfied or violated in the system analyzed. This process is guaranteed to terminate; in fact, it converges to an assumption that is necessary and sufficient for the property to hold in the specific system.

Our approach has been implemented in the Labeled Transition Systems Analyzer (LTSA) [27]. In this paper we discuss its application to the analysis of the Executive module of an experimental Mars Rover (K9) developed at NASA Ames. Note that our approach relies on standard features of model checkers, and as such it can be added in any such tool in a fairly straightforward way.

The remainder of the paper is organized as follows. We first provide some background in Section 2, followed by a high level description of the framework that we propose in Section 3. The algorithms that implement this framework are presented in Section 4. We discuss some theoretical and practical considerations for our framework in Section 5. Section 6 describes our experience with applying our approach to the Executive of the K9 Mars Rover and Section 7 presents some ideas for extending this work. Finally, Section 8 presents related work and Section 9 concludes the paper.

2 Background

The presentation of our approach is based on techniques for modeling and checking concurrent programs implemented in the LTSA tool [27]. The LTSA supports Compositional Reachability Analysis (CRA) of a software system based on its architecture, which, in general, has a hierarchical structure. CRA incrementally computes and abstracts the behavior of composite components based on the behavior of their immediate children in the hierarchy [17]. The flexibility that the LTSA provides in selecting any component in the hierarchy for analysis or abstraction makes it ideal for experimenting with our approach.

2.1 Labeled Transition Systems

The LTSA tool uses *labeled transition systems* (LTSs) to model the behavior of communicating components in a concurrent system. In the following, we present LTSs and the semantics of their operators in a typical process algebra style. However, note that our goal here is not to define a process algebra.

Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment. We use π to denote a special *error state*, which models the fact that a safety violation has occurred in the associated system. We require that the error state has no outgoing transitions because we are not interested in exploring states that follow a safety violation.

Formally, an LTS M is a four tuple $\langle Q, \alpha M, \delta, q_0 \rangle$ where:

- Q is a finite non-empty set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a set of observable actions called the *alphabet* of M
- $\delta \subseteq Q \times \alpha M \cup \{\tau\} \times Q$ is a transition relation
- $q_0 \in Q$ is the initial state

We use Π to denote the LTS $\langle \{\pi\}, \mathcal{A}ct, \emptyset, \pi \rangle$. An LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is *non-deterministic* if it contains τ-transitions or if $\exists (q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*.

Consider a simple communication channel that consists of two components whose LTSs are shown in Fig. 1. Note that the initial state of all LTSs in this paper is state 0. The *Input* LTS receives an input when the action `input` occurs, and then sends it to the *Output* LTS with action `send`. After some data is sent to it, *Output* produces output using the action `output` and acknowl-

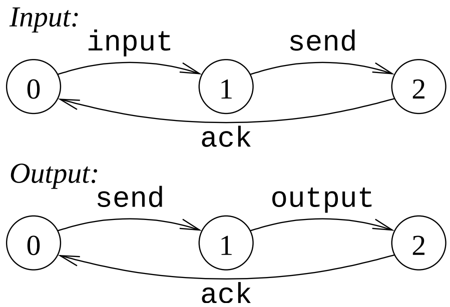

Fig. 1. Example LTSs

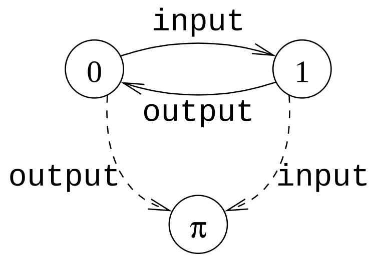

Fig. 2. *Order* Property

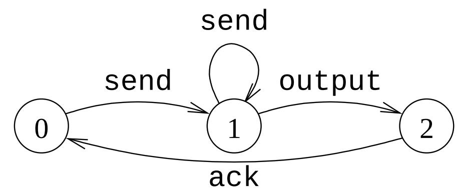

Fig. 3. LTS for *Output′*

edges that it has finished, by using the action `ack`. At this point, both LTSs return to their initial states so the process can be repeated.

2.2 Traces

A *trace* σ of an LTS M is a sequence of observable actions that M can perform starting at its initial state. For example, $\langle \texttt{input} \rangle$ and $\langle \texttt{input}, \texttt{send} \rangle$ are both traces of the *Input* LTS in Fig. 1. The set of all traces of M is called the *language* of M, denoted $\mathcal{L}(M)$. For $\Sigma \subseteq \mathcal{A}ct$, we use $\sigma \upharpoonright \Sigma$ to denote the trace obtained by removing from σ all occurrences of actions $a \notin \Sigma$.

2.3 Parallel Composition

Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q_0' \rangle$. We say that M *transits* into M' with action a, denoted $M \xrightarrow{a} M'$, if and only if $(q_0, a, q_0') \in \delta$ and either $Q = Q'$, $\alpha M = \alpha M'$, and $\delta = \delta'$ for $q_0' \neq \pi$, or, in the special case where $q_0' = \pi$, $M' = \Pi$.

The parallel composition operator $\|$ is a commutative and associative operator that combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions. For example, in the parallel composition of the *Input* and *Output* components from Fig. 1, actions `send` and `ack` will each be synchronized.

Formally, let $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$ be two LTSs. If $M_1 = \Pi$ or $M_2 = \Pi$, then $M_1 \parallel M_2 = \Pi$. Otherwise, $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined with the following transitional semantics, where a is either an observable action or τ:

- $$\frac{M_1 \xrightarrow{a} M_1',\ a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2}$$
- $$\frac{M_2 \xrightarrow{a} M_2',\ a \notin \alpha M_1}{M_1 \parallel M_2 \xrightarrow{a} M_1 \parallel M_2'}$$

- $$\frac{M_1 \xrightarrow{a} M_1',\, M_2 \xrightarrow{a} M_2',\, a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'}$$

2.4 Properties

We call a deterministic LTS that contains no π states a *safety LTS*. A *safety property* is specified as a safety LTS P, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP. An LTS M satisfies P, denoted as $M \models P$, if and only if $\forall \sigma \in \mathcal{L}(M)$ and $\forall \sigma' \in (\alpha M \cup \alpha P)^*$, if $\sigma' \upharpoonright \alpha M = \sigma$ then $\sigma' \upharpoonright \alpha P \in \mathcal{L}(P)$. Intuitively, this definition requires component M to satisfy property P irrespective of the environment in which M will be introduced.

When checking a property P, an *error LTS* denoted P_{err} is created, which traps possible violations with the π state. Formally, the error LTS of a property $P = \langle Q, \alpha P, \delta, q_0 \rangle$ is $P_{err} = \langle Q \cup \{\pi\}, \alpha P_{err}, \delta', q_0 \rangle$, where $\alpha P_{err} = \alpha P$ and

$$\delta' = \delta \cup \{(q, a, \pi) \mid a \in \alpha P \text{ and } \nexists q' \in Q : (q, a, q') \in \delta\}$$

Note that the error LTS is *complete*, meaning each state other than the error state has outgoing transitions for every action in its alphabet.

For example, the *Order* property shown in Fig. 2 captures a desired behavior of the communication channel shown in Fig. 1. The property comprises states $0, 1$ and the transitions denoted by solid arrows. It expresses the fact that `input`s and `output`s come in matched pairs, with the `input` always preceding the `output`. The dashed arrows illustrate the transitions to the error state that are added to the property to obtain its error LTS.

To detect violations of property P by component M, the parallel composition $M \parallel P_{err}$ is computed. It has been proved that M violates P if and only if the π state is reachable in $M \parallel P_{err}$ [7]. For example, state π is not reachable in $\mathit{Input} \parallel \mathit{Output} \parallel \mathit{Order}_{err}$, so we conclude that $\mathit{Input} \parallel \mathit{Output} \models \mathit{Order}$.

2.5 Assume-Guarantee Reasoning

In the assume-guarantee paradigm a formula is a triple $\langle A \rangle\ M\ \langle P \rangle$, where M is a component, P is a property, and A is an assumption about M's environment [30]. The formula is true if whenever M is part of a system satisfying A, then the system must also guarantee P.

The LTSA is particularly flexible in performing assume-guarantee reasoning. Both assumptions and properties are defined as safety LTSs[1]. In fact, a safety LTS A can be used as an assumption *or* as a property. To be used as an assumption for module M, A itself is composed with M, thus playing the role of an abstraction of M's environment. To be used as a property to be checked on M, A is turned into A_{err} and then composed with M.

To check an assume-guarantee formula $\langle A \rangle\ M\ \langle P \rangle$, where both A and P are safety LTSs, the LTSA computes $A \parallel M \parallel P_{err}$ and checks if state π is reachable

[1] Any LTS without π states can be transformed into a safety LTS by determinization [23].

in the composition. If it is, then $\langle A\rangle\ M\ \langle P\rangle$ is false, otherwise it is true.

2.6 Deterministic Finite State Automata and Safety LTSs

One of the components of our framework is a learning algorithm that produces deterministic finite state automata (DFAs), which our framework then uses as safety LTSs. A DFA M is a five tuple $\langle Q, \alpha M, \delta, q_0, F\rangle$ where $Q, \alpha M, \delta$, and q_0 are defined as for *deterministic* LTSs, and $F \subseteq Q$ is a set of accepting states.

For a DFA M and a string $\sigma \in (\alpha M)^*$, we use $\delta(q, \sigma)$ to denote the state that M will be in after reading σ starting at state q. A string σ is said to be *accepted* by a DFA $M = \langle Q, \alpha M, \delta, q_0, F\rangle$ if $\delta(q_0, \sigma) \in F$. The *language accepted by* M, denoted $\mathcal{L}(M)$ is the set $\{\sigma \mid \delta(q_0, \sigma) \in F\}$.

The DFAs returned by the learning algorithm in our context are *complete*, *minimal*, and *prefix-closed* (an automaton M is prefix-closed if $\mathcal{L}(M)$ is prefix-closed, i.e., for every $\sigma \in \mathcal{L}(M)$, every prefix of σ is also in $\mathcal{L}(M)$). These DFAs therefore contain a single non-accepting state. They can easily be transformed into safety LTSs by removing the non-accepting state, which corresponds to state π of an error LTS, and all transitions that lead into it.

3 Framework for Incremental Compositional Verification

Consider the case where a system is made up of two components, M_1 and M_2. As mentioned in the previous section, a formula $\langle A\rangle\ M\ \langle P\rangle$ is true if, whenever M is part of a system satisfying A, then the system must also guarantee property P. The simplest compositional proof rule shows that if $\langle A\rangle\ M_1\ \langle P\rangle$ and $\langle true\rangle\ M_2\ \langle A\rangle$ hold, then $\langle true\rangle\ M_1 \parallel M_2\ \langle P\rangle$ is true. This proof strategy can also be expressed as an inference rule as follows:

$$\frac{\begin{array}{ll}\text{(Step 1)} & \langle A\rangle\ M_1\ \langle P\rangle \\ \text{(Step 2)} & \langle \mathit{true}\rangle\ M_2\ \langle A\rangle\end{array}}{\langle \mathit{true}\rangle\ M_1 \parallel M_2\ \langle P\rangle}$$

Note that this rule is not symmetric in its use of the two components, and does not support circularity. Despite its simplicity, our experience with applying compositional verification to several applications has shown it to be a very useful rule in the context of checking safety properties. For the use of the compositional rule to be justified, the assumption must be more abstract than M_2, but still reflect M_2's behavior. Additionally, an appropriate assumption for the rule needs to be strong enough for M_1 to satisfy P in Step 1. Developing such an assumption is a non-trivial process.

To obtain appropriate assumptions, our framework applies the compositional rule in an iterative fashion as illustrated in Fig. 4. At each iteration i, an assumption A_i is provided based on some knowledge about the sys-

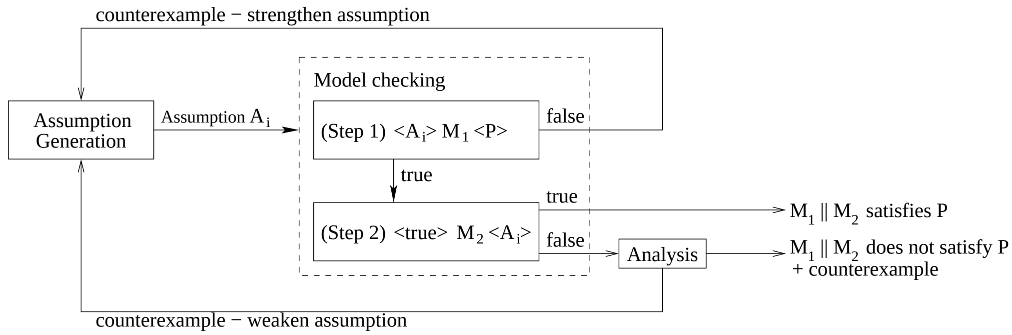

Fig. 4. Incremental compositional verification during iteration i

tem and on the results of the previous iteration. The two steps of the compositional rule are then applied. Step 1 is applied first, to check whether M_1 guarantees P in environments that satisfy A_i. If the result is false, it means that this assumption is too *weak*, i.e., A_i does not restrict the environment enough for P to be satisfied. The assumption therefore needs to be *strengthened* (which corresponds to removing behaviors from it) with the help of the counterexample produced by Step 1. In the context of the next assumption A_{i+1}, component M_1 should at least not exhibit the violating behavior reflected by this counterexample.

If Step 1 returns true, it means that A_i is strong enough for the property to be satisfied. To complete the proof, Step 2 must be applied to discharge A_i on M_2. If Step 2 returns true, then the compositional rule guarantees that P holds in $M_1 \parallel M_2$. If it returns false, further analysis is required to identify whether P is indeed violated in $M_1 \parallel M_2$ or whether A_i is stronger than necessary. Such analysis is based on the counterexample returned by Step 2. If A_i is too strong it must be *weakened* (i.e., behaviors must be added) in iteration $i + 1$. The result of such weakening will be that at least the behavior that the counterexample represents will be allowed by assumption A_{i+1}. The new assumption may of course be too weak, and therefore the entire process must be repeated.

To implement this iterative, incremental process in a fully automated way, our framework uses a learning algorithm for assumption generation and a model checker for the application of the two steps in the compositional rule. The learning algorithm is described in detail in the next section.

4 Algorithms

4.1 The L Algorithm*

The learning algorithm used by our approach was developed by Angluin [3] and later improved by Rivest and Schapire [31]. In this paper, we will refer to the *improved* version by the name of the original algorithm, L*. L* learns an unknown regular language and produces a

DFA that accepts it. Let U be an unknown regular language over some alphabet Σ. L* works by incrementally producing a sequence of candidate DFAs converging to the minimal DFA that accepts U. In order to learn U, L* needs to interact with a *Minimally Adequate Teacher*, from now on called *Teacher*. A Teacher must be able to correctly answer two types of questions from L*. The first type is a *membership query*, consisting of a string $\sigma \in \Sigma^*$; the answer is *true* if $\sigma \in U$, and *false* otherwise. The second type of question is a *conjecture*, in which L* gives to the Teacher a DFA C whose language it believes to be identical to U. The Teacher answers *true* if $\mathcal{L}(C) = U$. Otherwise the Teacher returns a counterexample, which is a string σ in the symmetric difference of $\mathcal{L}(C)$ and U.

At a higher level, L* creates a table where it incrementally records whether strings in Σ^* belong to U. It does this by making membership queries to the Teacher. At various stages L* decides to make a conjecture. It constructs a candidate automaton C based on the information contained in the table and asks the Teacher whether the conjectured automaton is correct. If it is, the algorithm terminates. Otherwise, L* uses the counterexample returned by the Teacher to extend the table with strings that witness differences between $\mathcal{L}(C)$ and U.

In the following more detailed presentation of the algorithm, line numbers refer to L*'s illustration in Fig. 5. L* builds an observation table (S, E, T) where S and E are a set of prefixes and suffixes, respectively, both over Σ^*. In addition, T is a function mapping $(S \cup S \cdot \Sigma) \cdot E$ to $\{\text{true}, \text{false}\}$, where the operator "$\cdot$" is defined as follows. Given two sets of event sequences P and Q, $P \cdot Q = \{pq \mid p \in P \text{ and } q \in Q\}$, where pq represents the concatenation of the event sequences p and q. Initially, L* sets S and E to $\{\lambda\}$ (line 1), where λ represents the empty string. Subsequently, it updates the function T by making membership queries so that it has a mapping for every string in $(S \cup S \cdot \Sigma) \cdot E$ (line 2). It then checks whether the observation table is *closed*, i.e., whether

$$\forall s \in S, \forall a \in \Sigma, \exists s' \in S, \forall e \in E : T(sae) = T(s'e)$$

If (S, E, T) is not closed, then sa is added to S where $s \in S$ and $a \in \Sigma$ are the elements for which there is no $s' \in S$ (line 3). Once sa has been added to S, T needs to be updated (line 4). Lines 3 and 4 are repeated until (S, E, T) is closed.

Once the observation table is closed, a candidate DFA $C = \langle Q, \alpha C, \delta, q_0, F \rangle$ is constructed (line 5), with states $Q = S$, initial state $q_0 = \lambda$, and alphabet $\alpha C = \Sigma$, where Σ is the alphabet of the unknown language U. The set F consists of the states $s \in S$ such that $T(s) = \text{true}$. The transition relation δ is defined as $\delta(s, a) = s'$ where $\forall e \in E : T(sae) = T(s'e)$. Such an s' is guaranteed to exist when (S, E, T) is closed. Next, L* conjectures that C is correct (line 6). If the conjectured automaton is correct, i.e., if $\mathcal{L}(C) = U$, L* returns C as correct (line 7),

(1) let $S = E = \{\lambda\}$
loop {
(2)   update T using queries
  while (S, E, T) is not closed {
(3)   add sa to S to make S closed where $s \in S$ and $a \in \Sigma$
(4)   update T using queries
  }
(5)   construct candidate DFA C from (S, E, T)
(6)   conjecture that C is correct
(7)   if C is correct return C
  else
(8)   add $e \in \Sigma^*$ that witnesses the counterexample to E
}

Fig. 5. The L* Algorithm

otherwise it receives a counterexample $c \in \Sigma^*$ from the Teacher.

The counterexample c is analyzed by L* to find a suffix e of c that witnesses a difference between $\mathcal{L}(C)$ and U; e must be such that adding it to E will cause the next conjectured automaton to reflect this difference[2] (line 8). Once e has been added to E, L* iterates the entire process by looping around to line 2.

4.1.1 Characteristics of L*

L* is guaranteed to terminate with a minimal automaton M for the unknown language U. Moreover, for each closed observation table (S, E, T), the candidate DFA C that L* constructs is smallest, in the sense that any other DFA consistent[3] with the function T has at least as many states as C. This characteristic of L* makes it particularly attractive for our framework. Our framework uses L* to generate assumptions that are then used by a model checker to apply the two steps of the compositional rule. Therefore smaller assumptions may contribute to smaller state spaces. The conjectures made by L* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than M. Therefore, if M has n states, L* makes at most $n-1$ incorrect conjectures. The number of membership queries made by L* is $\mathcal{O}\left(kn^2 + n \log m\right)$, where

[2] The procedure for finding e is beyond the scope of this paper, but is described in [31].

[3] A DFA C is consistent with function T if, for every σ in $(S \cup S \cdot \Sigma) \cdot E$, $\sigma \in \mathcal{L}(C)$ if and only if $T(\sigma) = \text{true}$.

k is the size of the alphabet of U, n is the number of states in the minimal DFA for U, and m is the length of the longest counterexample returned when a conjecture is made.

4.2 Learning for Assume-Guarantee Reasoning

Assume a system $M_1 \parallel M_2$, and a property P that needs to be satisfied in the system. In the context of the compositional rule presented in Section 3, the learning algorithm is called to guess an assumption that can be used in the rule to prove or disprove P. An assumption with which the rule is guaranteed to return conclusive results is the *weakest assumption* A_w under which M_1 satisfies P. Assumption A_w describes exactly those traces over $\Sigma = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$ in the context of which M_1 satisfies P. Intuitively, A_w is similar to a controller (the most general one) in the context of which M_1 satisfies P. The language $\mathcal{L}(A_w)$ of the assumption contains at least *all* traces of M_2 abstracted to Σ that prevent M_1 from violating P. Formally [18], A_w is such that, for any environment component M_E:

$$\langle true \rangle\ M_1 \parallel M_E\ \langle P \rangle \text{ if and only if } \langle true \rangle\ M_E\ \langle A_w \rangle$$

In our framework, L* learns the traces of A_w through the iterative process described in Section 3. The process terminates as soon as compositional verification returns conclusive results, which is often before the weakest assumption A_w is computed by L*. For L* to learn A_w, we need to provide a Teacher that is able to answer the two different kinds of questions that L* asks. Our approach uses *model checking* to implement such a Teacher.

4.2.1 Membership Queries

To answer a membership query for $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ in Σ^* the Teacher simulates the query on $M_1 \parallel P_{err}$. For clarity of presentation we will reduce such simulations to model checking, although we have implemented them more efficiently, directly as simulations. So for string σ, the Teacher first builds $A_\sigma = \langle Q, \alpha A_\sigma, \delta, q_0 \rangle$ where $Q = \{q^0, q^1, \ldots, q^n\}$, $\alpha A_\sigma = \Sigma$, $\delta = \{(q^i, a^{i+1}, q^{i+1}) \mid 0 \leq i < n\}$, and $q_0 = q^0$. The Teacher then model checks the assume-guarantee triple $\langle A_\sigma \rangle\ M_1\ \langle P \rangle$, as described in section 2.5. If true is returned, it means that $\sigma \in \mathcal{L}(A_w)$, because M_1 does not violate P in the context of σ, so the Teacher returns true. Otherwise, the answer to the membership query is false.

4.2.2 Conjectures

Due to the fact that in our case the language $\mathcal{L}(A_w)$ that is being learned is prefix-closed, all conjectured automata returned by L* are also prefix-closed. Our framework transforms these conjectured automata into safety LTSs (see Section 2), which constitute the intermediate assumptions A_i. In our framework, the first priority is to guide L* towards an assumption that is strong enough to make Step 1 of the compositional rule return true. Once this is accomplished, the resulting assumption may be too strong, in which case our framework guides L* towards an assumption that is weak enough to make Step 2 return conclusive results about whether the system satisfies P. The way the Teacher that we have implemented reflects this approach is by using two oracles and counterexample analysis to answer conjectures as follows.

Oracle 1 performs Step 1 in Fig. 4, i.e., it checks $\langle A_i \rangle\ M_1\ \langle P \rangle$. If this does not hold, the model checker returns a counterexample c and the Teacher informs L* that the conjectured automaton A_i is not correct and provides $c \upharpoonright \Sigma$ to witness this fact. If, instead, $\langle A_i \rangle\ M_1\ \langle P \rangle$ holds, the Teacher forwards A_i to Oracle 2.

Oracle 2 performs Step 2 in Fig. 4 by checking $\langle true \rangle\ M_2\ \langle A_i \rangle$. If the result of model checking is true, our framework terminates the verification. A_i may not be the weakest assumption, but since, according to the compositional rule, P has been proved on $M_1 \parallel M_2$, it is not necessary to continue the learning process. On the other hand, if model checking returns a counterexample, the Teacher performs some analysis to determine the underlying reason (see Section 3 and Fig. 4).

Counterexample analysis is performed by the Teacher in a way similar to that used for answering membership queries. Let c be the counterexample returned by Oracle 2. The Teacher computes $A_{c\upharpoonright\Sigma}$ and checks $\langle A_{c\upharpoonright\Sigma} \rangle\ M_1\ \langle P \rangle$. If true, the Teacher returns false; although A_i is strong enough to prevent M_1 from violating the property, it is actually too strong, since M_1 does not violate P in the context of c, and the Teacher returns $c \upharpoonright \Sigma$ as a counterexample for the conjectured assumption A_i. If the model checker returns false with some counterexample c', it means that P is violated in $M_1 \parallel M_2$. To generate a counterexample for $\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$ our framework composes c and c' in a way similar to the parallel composition of LTSs: common actions in c and c' are synchronized and some interleaving instance of the remaining actions is selected.

4.3 Example

Given components *Input* and *Output* as shown in Fig. 1 and the property *Order* shown in Fig. 2, we will check $\langle true \rangle$ *Input* $\parallel$ *Output* $\langle Order \rangle$ by using our approach. The alphabet of the assumptions that will be used in the compositional rule is $\Sigma = ((\alpha Input \cup \alpha Order) \cap \alpha Output) = \{\mathtt{send}, \mathtt{output}, \mathtt{ack}\}$.

As described, at each iteration L* updates its observation table and produces a candidate assumption whenever the table becomes closed. The first closed table obtained is shown in Table 1 and its associated assumption, A_1, is shown in Fig. 6. The Teacher answers this conjecture by first invoking Oracle 1, which checks $\langle A_1 \rangle$ *Input* $\langle Order \rangle$. Oracle 1 returns false, with counterexample $\sigma = \langle \mathtt{input}, \mathtt{send}, \mathtt{ack}, \mathtt{input} \rangle$, which describes a trace in $A_1 \parallel Input \parallel Order_{err}$ that leads to state π.

The Teacher therefore returns counterexample $\sigma \upharpoonright \Sigma = \langle \mathtt{send}, \mathtt{ack} \rangle$ to L*, which uses queries to update its observation table until it is closed. From this table, shown in Table 2, the assumption A_2, shown in Fig. 7, is

Table 1. Mapping T_1

		E_1
	T_1	λ
S_1	λ	true
	`output`	false
$S_1 \cdot \Sigma$	`ack`	true
	`output`	false
	`send`	true
	`output`, `ack`	false
	`output`, `output`	false
	`output`, `send`	false

Table 2. Mapping T_2

		E_2	
	T_2	λ	`ack`
S_2	λ	true	true
	`output`	false	false
	`send`	true	false
$S_2 \cdot \Sigma$	`ack`	true	true
	`output`	false	false
	`send`	true	false
	`output`, `ack`	false	false
	`output`, `output`	false	false
	`output`, `send`	false	false
	`send`, `ack`	false	false
	`send`, `output`	true	true
	`send`, `send`	true	true

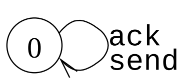

Fig. 6. A_1

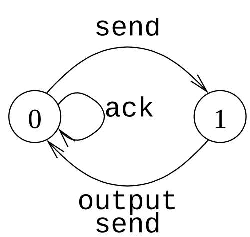

Fig. 7. A_2

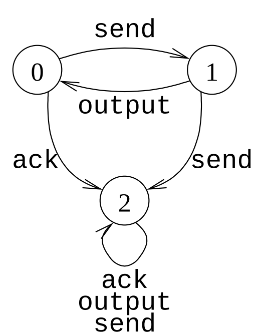

Fig. 8. A_3

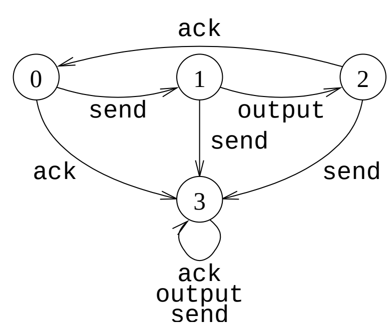

Fig. 9. A_4

constructed and conjectured to the Teacher. This time, Oracle 1 reports that $\langle A_2 \rangle$ *Input* $\langle Order \rangle$ is true, meaning the assumption is not too weak. The Teacher calls Oracle 2 to determine if $\langle true \rangle$ *Output* $\langle A_2 \rangle$. This is also true, so our algorithm reports that $\langle true \rangle$ *Input* $\|$ *Output* $\langle Order \rangle$ holds.

This example did not involve weakening of the assumptions produced by L*, since the assumption A_2 was sufficient for the compositional proof. This will not always be the case. For example, let us substitute *Output* by *Output′* illustrated in Fig. 3, which allows multiple `send` actions to occur before producing `output`. The verification process will be identical to the previous case, until Oracle 2 is invoked by the Teacher when it conjectures that A_2 is correct. Oracle 2 returns that $\langle true \rangle$ $Output'$ $\langle A_2 \rangle$ is false, with counterexample $\langle \mathtt{send}, \mathtt{send}, \mathtt{output} \rangle$. The Teacher analyzes this counterexample and determines that in the context of this trace, *Input* does not violate *Order*. The trace is returned to L*, which will weaken the conjectured assumption. The process involves two more iterations, during which assumptions A_3 (Fig. 8) and A_4 (Fig. 9), are conjectured. Using A_4, which is the weakest assumption A_w, both Oracles report true, so our framework reports that $\langle true \rangle$ $Input \parallel Output'$ $\langle Order \rangle$ holds.

5 Discussion

5.1 Correctness and Termination

Lemma 1. *The Teacher in our framework correctly answers L* membership queries and conjectures for the language of the weakest assumption $\mathcal{L}(A_w)$.*

Proof. The language of A_w describes exactly those traces σ over $\Sigma = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$ in the context of which M_1 satisfies P, i.e. for which $\langle A_\sigma \rangle$ M_1 $\langle P \rangle$ holds.

As mentioned, the Teacher answers membership queries by checking $\langle A_\sigma \rangle$ M_1 $\langle P \rangle$ and therefore it answers them correctly.

Our framework only uses false answers to the conjectures presented to the Teacher, and we therefore argue correctness only in terms of negative replies. We will show that whenever the Teacher replies false to conjectures, it does this correctly. There are two instances where the Teacher replies false:

1. When Oracle 1 finds that $\langle A_i \rangle$ M_1 $\langle P \rangle$ is false. By definition, $\langle A_w \rangle$ M_1 $\langle P \rangle$ holds, and therefore A_i can not be A_w.
2. During counterexample analysis, when for the counterxample c produced by Oracle 2, $\langle A_{c\restriction\Sigma} \rangle$ M_1 $\langle P \rangle$ holds. Therefore, by the definition of the language of A_w, $c \restriction \Sigma \in \mathcal{L}(A)_w$. However, based on Step 2 as performed by Oracle 2, c is a trace in M_2 such that $c \restriction \Sigma$ does not belong to $\mathcal{L}(A_i)$. As a result, $c \restriction \Sigma$ is a trace in $\mathcal{L}(A_w)$ that is not in $\mathcal{L}(A_i)$.

□

Theorem 1. *Given components M_1 and M_2, and property P, the algorithm implemented by our framework terminates and it returns true if P holds on $M_1 \parallel M_2$ and false otherwise.*

Proof. To prove the theorem we will first argue correctness and then termination of our approach.

Correctness: Our framework only returns true when both steps of the compositional rule return true, and therefore correctness is guaranteed by the compositional rule. Our framework reports an error when it detects a trace σ of M_2 which, when simulated on M_1, violates the property, which implies that $M_1 \parallel M_2$ violates P.

Termination: At any iteration, our algorithm returns true or false and terminates, or continues by providing a counterexample to L*. By correctness of L* and from Lemma 1, we are guaranteed that if it keeps receiving counterexamples, it will eventually, at some iteration i, produce A_w. During this iteration, Step 1 will return true by definition of A_w. The Teacher will therefore apply Step 2, which will return either true and terminate, or a counterexample. This counterexample represents a trace of M_2 that is not contained in $\mathcal{L}(A_w)$. Since, as discussed in Section 4, A_w is both necessary and sufficient, analysis of the counterexample will return false, and the algorithm will terminate. □

5.2 Practical Considerations

In our context, the languages queried by L* are prefix-closed. This is because our technique applies to purely safety properties; any finite prefix of a trace that satisfies such a property must also satisfy the property. Therefore, when for some string σ a membership query $\langle A_\sigma \rangle\ M_1\ \langle P \rangle$ returns false, we know that for any extension of σ the answer will also be false. We can thus improve the efficiency of the algorithm by reducing the cost of some of the membership queries that are answered by the Teacher. For example, in the observation table shown in Table 1, the entry for $\langle \texttt{output} \rangle$ is false. The Teacher can return false for the queries $\langle \texttt{output}, \texttt{ack} \rangle$, $\langle \texttt{output}, \texttt{send} \rangle$, and $\langle \texttt{output}, \texttt{output} \rangle$ without invoking the model checker.

In our framework, membership queries, conjectures, and counterexample analysis all involve model checking, which is performed on-the-fly. The assumptions that are used in these steps are increasing in size, and grow no larger than the size of A_w, i.e. $|A_i| < |A_{i+1}|$ and $|A_i| \leq |A_w|$. However, we should note that there is no monotonicity at the semantic level, i.e. it is not necessarily the case that $\mathcal{L}(A_i) \subseteq \mathcal{L}(A_{i+1})$. This is the reason why our framework requires both strengthening and weakening of the assumptions.

In our experience, for well-designed systems, the interfaces between components are small. Therefore, assumptions are expected to be significantly smaller than the environment that they represent in the compositional rules. Moreover, the controllability information that we use to derive these assumptions, and the fact that we take the properties into account in building them, typically allow us to achieve further reduction than abstraction techniques that would merely simplify models based on component interfaces. Although L* needs to maintain an observation table, this table does not need to be kept in memory while the model checking is performed.

Note that our framework provides an anytime approach [13] to compositional verification. If memory is not sufficient to reach termination, intermediate assumptions are generated, which may be useful in approximating the requirements that a component places on its environment to satisfy certain properties.

6 Experience

We implemented the assume-guarantee framework described above in the LTSA tool. In this section, we will describe its application to the analysis of a design-level model of the executive subsystem for the K9 Mars Rover controller developed at NASA Ames.

6.1 K9 Mars Rover Executive

The executive receives flexible plans from a planner, which it executes according to the plan language semantics. A plan is a hierarchical structure of actions that the Rover must perform. Traditionally, plans are deterministic sequences of actions. However, increased Rover autonomy requires added flexibility. The plan language therefore allows for branching based on state or temporal conditions that need to be checked, and also for flexibility with respect to the starting time of an action. The plan language allows the association of each action with a number of state or temporal pre-, maintenance, and post-conditions, which must hold before, during, and on completion of the action execution, respectively.

The executive has been implemented as a multi-threaded system (see Fig. 10), made up of a main coordinating component named *Executive*, components for monitoring the state conditions *ExecCondChecker*, and temporal conditions *ExecTimerChecker* – each further decomposed into two threads – and finally an *ActionExecution* thread that is responsible for issuing the commands to the Rover. Synchronization between these threads is performed through mutexes and condition variables.

In their description of the system design, the developers explicitly communicated to us their intentions as to which mutexes were to protect accesses to which shared variables. They also provided some design documents that described the synchronization between components in an ad-hoc flowchart-style language. The descriptions looked very much like LTSs, which allowed us to translate them in a straightforward and systematic, albeit manual, way into the input language of the LTSA.

The first properties that we checked on the model thus created was whether each access to a shared variable between threads was protected by the appropriate mutex. We then proceeded to a more elaborate coordination property that was provided to us by the developers, the verification of which we describe in detail in this section. The property refers to a subsystem of the executive consisting of the *Executive* and the *ExecCondChecker*, and states the following (see Fig. 10):

> "for the variable *savedWakeUpStruct* of the *ExecCondChecker* shared with the *Executive*, if the *Executive* reads the value of the variable, then the

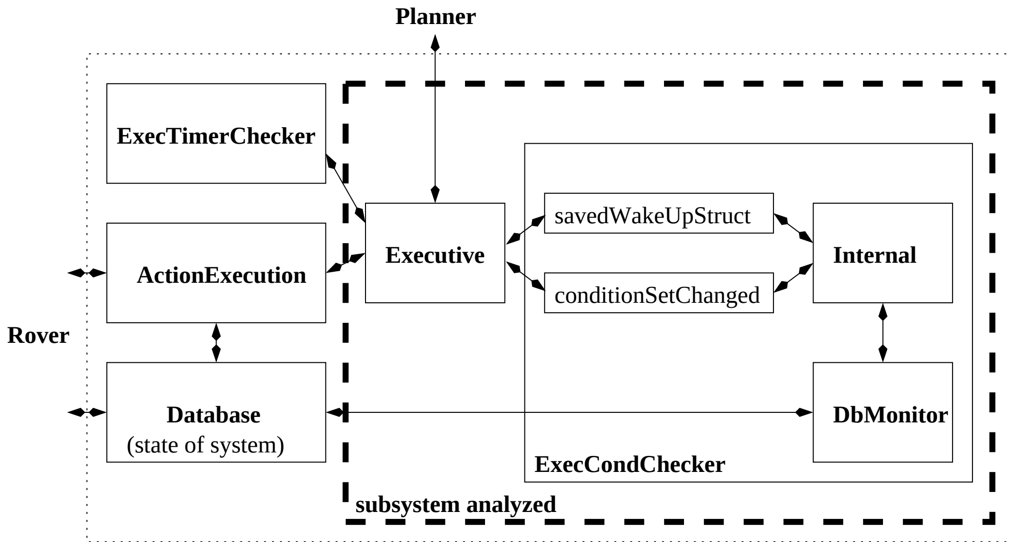

Fig. 10. The Executive of the K9 Mars Rover

ExecCondChecker should not read this value before the *Executive* clears it first."

6.2 K9 Rover Analysis

For compositional verification of the above property, we set M_1 = *ExecCondChecker* and M_2 = *Executive*. We compared the performance of our learning framework to two alternative approaches for checking this property. The first is what we call the "monolithic" approach; this is the standard non-compositional approach that checks the property on *Executive* $\|$ *ExecCondChecker*. The second approach [18] is compositional but not iterative. It is also based on the simple compositional rule presented in Section 3. It first generates the weakest assumption A_w for Step 1 of the rule to be true, and then applies Step 2 of the rule, i.e. it discharges the assumption. The construction of the weakest assumption is performed in three stages during which several transformations are applied to $M_1 \parallel P_{err}$.

The experiment was conducted on an Apple G4 867 MHz with 384 Mb of memory running OS X 10.3 using Java SDK version 1.4.1_01. We report the following results. For the learning framework, we describe the size (in terms of numbers of states) of the candidate assumptions that it generates, as well as the size (in terms of number of states and transitions) of the state space that each oracle needs to explore. The three approaches are then compared in terms of the total time needed to perform the verification and the size of the largest state space explored. For the learning framework, this is the largest state space among the ones that are explored by the oracles. For the approach based on the weakest assumption, this is the largest state space between 1) $M_1 \parallel P_{err}$ that needs to be explored for the generation

Table 3. Results of Learning for the Rover Example

Iteration	$\lvert A_i \rvert$	States	Transitions	Result
1 - Oracle 1	1	294	1,548	Not too weak
1 - Oracle 2	1	16	93	Too strong
2 - Oracle 1	2	269	1,560	Too weak
3 - Oracle 1	3	**541**	**3,066**	Too weak
4 - Oracle 1	5	12	69	Too weak
5 - Oracle 1	6	474	2,706	Not too weak
5 - Oracle 2	6	32	197	Property does not hold

Table 4. Comparison of Approaches for the Rover Example

Approach	$\lvert A \rvert$	Max. States	Max. Transitions	Time (s)
Monolithic	N/A	3,630	34,653	0.677
Generate and Discharge A_w	6	588	4,272	101.574
Learning	6	541	3,066	6.696

of the assumption, and 2) the state space that is explored by the discharge of the assumption.

The results of the learning framework are illustrated in Table 3. The $|A_i|$ column gives the number of states of the assumptions generated. The entries in the States and Transitions columns that are in bold face indicate the maximum value for those columns. The table also shows the number of states and transitions explored during the analysis of the assumption. In iteration 1, Oracle 1 determined that the conjectured assumption was not too weak, so the conjectured assumption was given to Oracle 2, which returned a counterexample. When simulated on the *ExecCondChecker*, this counterexample did not lead to an error state, indicating that the conjectured assumption was too strong. In iterations 2-4, Oracle 1 determined that the conjectured assumption was too weak. In iteration 5, the conjectured assumption was not too weak and given to Oracle 2, which returned a counterexample. This counterexample, when simulated on the *ExecCondChecker*, led to an error state. The analysis therefore concluded that the property does not hold.

The largest state space involved in the application of our approach was explored by Oracle 1 during iteration 3, and consisted of 541 states and 3,066 transitions. Running the learning algorithm required 6.696 seconds. Table 4 gives the results of the learning framework as compared to the other two approaches we tried. For the two compositional approaches, the table gives the size of the assumption used to show that the property does not hold. Both compositional approaches explore almost an order of magnitude fewer states and transitions than the monolithic approach, with the learning approach performing slightly better. However, the two compositional approaches take longer than the monolithic approach, with the learning framework requiring less time than the one that generates and discharges A_w[4].

6.3 Corrected Rover

The analysis performed on the Rover uncovered an error in its design. The counterexample obtained described a scenario where, after reading *savedWakeUpStruct* and before clearing it, the *Executive* would perform a *wait* that released the mutex associated with *savedWakeUpStruct*. It is then possible for the *ExecCondChecker* to read the same value, and thus violate the property. We corrected this problem by getting the *Executive* to clear *savedWakeUpStruct* before releasing the mutex.

To make sure that the change we applied corrects the problem, we verified the subsystem again. Table 5 and Table 6 give the results obtained for the corrected Rover. The performance of the three approaches on the corrected Rover is similar to their performance on the buggy Rover. Again, the compositional approaches explored fewer states and transitions than the monolithic approach, but required more time, with the learning framework performing better than the one that generates and dischargesA_w.

[4] Generating A_w took most of the time, requiring 99.719 seconds.

In the above experiments, the compositional approaches outperform the monolithic approach in terms of sizes of state spaces that they need to explore, but are outperformed in terms of the time consumed in carrying out the verification. For the learning framework, the latter is due to the iterative learning of assumptions. For the approach based on generating the weakest assumption, it is due to a minimization step involved in the assumption generation process.

In general, we believe that the potential benefits of our approach in terms of memory outweigh the time overhead that it may incur. Space complexity is the major concern in model checking, so approaches that provide automated support for decomposing a model checking problem into smaller ones (in terms of space) are crucial for achieving the verification of realistically-sized systems.

7 Extensions

7.1 Starting with a supplied assumption

Determining an assumption to complete an assume-guarantee proof can be a difficult task. The framework presented learns an assumption to complete an assume-guarantee proof automatically with no input from the analyst. However, analysts typically have knowledge about the system they are verifying and can use this

Table 5. Results of Learning for the Corrected Rover Example

Iteration	$\|A_i\|$	States	Transitions	Result
1 - Oracle 1	1	294	1,548	Not too weak
1 - Oracle 2	1	15	90	Too strong
2 - Oracle 1	2	269	1,560	Too weak
3 - Oracle 1	3	**541**	**3,066**	Too weak
4 - Oracle 1	5	12	69	Too weak
5 - Oracle 1	6	474	2,706	Not too weak
5 - Oracle 2	6	105	653	Property holds

Table 6. Comparison of Approaches for the Corrected Rover Example

Approach	$\|A\|$	Max. States	Max. Transitions	Time (s)
Monolithic	N/A	4,672	44,464	0.574
Generate and Discharge A_w	6	588	4,272	100.034
Learning	6	541	3,066	6.044

knowledge to formulate assumptions. If we can adapt our algorithms to make use of these analyst-supplied assumptions, we should be able to improve the efficiency of our approach.

For example, consider the case where an analyst has developed an assumption A_1, that he or she believes to be correct. In this case, the first thing that needs to be checked is if $M_2 \models A_1$. If this is false, then the assumption supplied by the analyst is incorrect, and this should be reported. If $M_2 \models A_1$, then the next thing to be checked is if $M_1 \parallel A_1 \models P$. If this is true, then the assumption A_1 is correct and sufficient to complete the assume-guarantee proof. Otherwise, the assumption A_1 is not sufficient, and our learning algorithm can be used to learn an assumption A_2 such that $\langle A_2 \rangle\ M_1 \parallel A_1\ \langle P \rangle$ and $\langle true \rangle\ M_2\ \langle A_2 \rangle$.

As mentioned earlier, for the Rover executive the developers expressed to us their intentions as to which mutexes were to protect accesses to which shared variables. In the analysis of the Rover we can therefore introduce an assumption stating that the *Executive* only accesses *savedWakeUpStruct* when it holds the appropriate mutex. We repeated the verification experiments with this additional assumption, the results of which are shown in Tables 7 and 8. The additional assumption is not introduced in the monolithic approach since the behavior of both components is already included. For the compositional approaches we had to also show that the *Executive*

Table 7. Results of Learning for the Corrected Rover Example with Additional Assumption

Iteration	$\lvert A_i \rvert$	States	Transitions	Result
1 - Oracle 1	1	5	24	Too weak
2 - Oracle 1	2	268	1,408	Too weak
3 - Oracle 1	3	235	1,209	Too weak
4 - Oracle 1	5	**464**	**2,500**	Not too weak
4 - Oracle 2	5	97	609	Property holds

Table 8. Comparison of Approaches for the Corrected Rover Example with Additional Assumption

Approach	$\lvert A \rvert$	Max. States	Max. Transitions	Time (s)
Monolithic	N/A	4,672	44,464	0.677
Generate and Discharge A_w	6	544	3,082	19.139
Learning	5	464	2,500	4.575
Discharge Additional Assumption	2	41	252	0.332

actually satisfies the additional assumption. The cost of this step is reported in the last row of Table 8.

Using this additional assumption improved the performance of both compositional approaches, particularly the running time of generating A_w, which now required 18.790 seconds compared to 99.719 seconds without the additional assumption. As before, the compositional approaches explore fewer states and transitions than the monolithic approach, but require more time, with the learning approach performing better than the approach the generates and then discharges A_w.

In the Rover case study, the use of an assumption provided by the developer/analyst improved the performance of the learning algorithm. However, we do not always expect that the analyst will be able to supply an assumption A_1 such that $M_2 \models A_1$. The approach presented above cannot deal with this case. The assumption A_1 might encode behavior the analyst believes to be true, so, even if it is incorrect, it should provide a reasonable starting point for learning a correct assumption. To address this, we could adapt the approach presented by Groce et al. in [19], which improves the L* learning algorithm by using a (possibly incorrect) supplied automaton to complete initial values for the sets S and E. In the experiments from [19], their approach was shown to save both time and memory over starting with $S = E = \{\lambda\}$. However, their work uses the Angluin version of L* rather than the Rivest and Schapire version. We wish to investigate if we could obtain similar benefits in our framework, especially since we use the Rivest and

Shapire version, which has better worst-case complexity than the Angluin version.

7.2 Generalization

Our approach has been presented in the context of two components. Assume now that a system consists of n components $M_1 \parallel \cdots \parallel M_n$. The simplest way to generalize our approach is to group these components into two higher level components, and apply the compositional rules as already discussed. Another possibility is to handle the general case without computing the composition of any components directly. Our algorithm provides a way of checking $\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$ in a compositional way. If M_2 consists of more than one component, our algorithm could be applied recursively for Step 2. This is an interesting future direction, in particular since the membership queries concentrate on a single component at a time. However, we need to further investigate how meaningful such an approach would be in practice.

7.3 Computing the Weakest Assumption

L* can also be used to learn the weakest possible assumption A_w that will prevent a component M_1 from violating a property P. This assumption will be generated without knowing M_2, the component M_1 interacts with. The only place in our assume-guarantee framework where M_2 is used is in Oracle 2, when the Teacher tries to determine if the Assumption generated is too strong. Oracle 2 can be replaced by a conformance checker, for

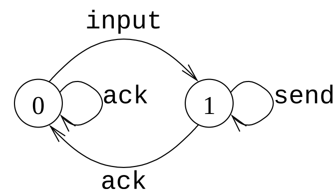

Fig. 11. A_5

example the W-Method [8], which is designed to expose a difference between a specification and an implementation. This will produce a set of sequences that are guaranteed to expose an error in the conjectured assumption if one exists. The sequence of intermediate assumptions conjectured by the Teacher are approximate and become more refined the longer L* runs.

The W-Method is an exponential algorithm. As a result, we would expect this approach to be used only when other approaches to assumption generation, such as [18], run out of memory. Additionally, the execution of one of the algorithm's loops requires knowledge about the size of M_1, resulting in a very expensive computation. However, this is still an anytime algorithm and can be used to generate a sequence of approximate assumptions.

7.4 Symmetric rules

The assume-guarantee rule that we use in our framework is not symmetric in its use of the two components. This fact makes our framework sensitive to the order in which the components are checked.

Consider again the example from Section 4.3, where the *Output* component is replaced by *Output'*. For the case were $M_1 = \mathit{Input}$, $M_2 = \mathit{Output}'$, and $P = \mathit{Order}$, our learning framework concludes that $\langle true \rangle$ *Input* $\|$ $Output'$ $\langle Order \rangle$ holds and it produces the assumption A_4, shown in Fig. 9. If we change the order in which the two components are considered by our learning framework, i.e. we set $M_1 = Output'$ and $M_2 = Input$, then the framework again concludes that $\langle true \rangle$ *Input* $\|$ $Output'$ $\langle Order \rangle$ holds and it produces the (final) assumption A_5, shown in Fig. 11. This assumption is obtained after just two (rather than 4) iterations and is smaller than A_4. Therefore, how LTS components of a system get assigned to M_1 and M_2 can have a significant impact on the efficiency of the learning algorithm.

Several other assume-guarantee rules exist in the literature that are symmetric and involve circularity, e.g., [5]. Symmetric rules use assumptions for each component of the system. We are interested in incorporating such rules into our framework, as we expect that their use will lead to earlier termination of the iterative process and to even smaller assumptions; we have done preliminary work in this direction [4].

8 Related Work

One way of addressing both the design and verification of large systems is to use their natural decomposition into components. Formal techniques for support of component-based design are gaining prominence, see for example [11, 12]. In order to reason formally about components in isolation, some form of assumption (either implicit or explicit) about the interaction with, or interference from, the environment has to be made. Even though we have sound and complete reasoning systems for assume-guarantee reasoning, see for example [9, 20, 25, 30, 33], it is always a mental challenge to obtain the most appropriate assumption [22].

It is even more of a challenge to find automated techniques to support this style of reasoning. The thread modular reasoning underlying the Calvin tool [15] is one start in this direction. In the framework of temporal logic, the work on Alternating-time Temporal Logic ATL [1] was proposed for the specification and verification of open systems together with automated support via symbolic model checking procedures. The Mocha toolkit [2] provides support for modular verification of components with requirement specifications based on the ATL.

Recently, Henzinger et al. [21] have presented a framework for thread-modular abstraction refinement, in which assumptions and guarantees are both refined in an iterative fashion. The framework applies to programs that communicate through shared variables, and, unlike our approach where assumptions are controllers of the component that is being analyzed, the assumptions in [21] are abstractions of the environment components. The work of Flanagan and Qadeer also focuses on a shared-memory communication model [16], but does not address notions of abstractions as is done in [21] . Jeffords and Heitmeyer use an invariant generation tool to generate invariants for components that can be used to complete an assume-guarantee proof [24]. While their proof rules are sound and complete, their invariant generation algorithm is not guaranteed to produce invariants that will complete an assume-guarantee proof even if such invariants exist.

In previous work [18], we presented an algorithm for automatically generating the weakest possible assumption for a component to satisfy a required property. Although the motivation of that work is different, the ability to generate the weakest assumption can also be used to automate assume-guarantee reasoning. The algorithm in [18] does not compute partial results, meaning no assumption is obtained if the computation runs out of memory. This may happen if the state space of the component is too large. The approach presented here generates assumptions incrementally and may terminate before A_w is computed. Moreover, even if it runs out of memory before reaching conclusive results, intermediate assumptions may be used to give some indication to the developer of the requirements that the component places on its environment.

The problem of generating an assumption for a component is similar to the problem of generating component interfaces to deal with intermediate state explosion in CRA. Several approaches have been defined for automatically abstracting a component's environment to obtain interfaces [6, 26]. These approaches do not address the issue of incrementally refining interfaces, as needed for carrying out an assume-guarantee proof.

Learning in the context of model checking has also been investigated in [19], but with a different goal. In that work, the L* Algorithm is used to generate a model of a software system which can then be fed to a model checker. A conformance checker determines if the model accurately describes the system.

9 Conclusions

Although theoretical frameworks for sound and complete assume-guarantee reasoning have existed for decades, their practical impact has been limited because they involve non-trivial human interaction. In this paper, we presented a novel approach to performing such reasoning in a fully automatic fashion. Our approach uses a learning algorithm to generate and refine assumptions based on queries and counterexamples, in an iterative process. The process is guaranteed to terminate, and return true if a property holds in a system, and a counterexample otherwise. If memory is not sufficient to reach termination, intermediate assumptions are generated, which may be useful in approximating the requirements that a component places on its environment to satisfy certain properties.

One advantage of our approach is its generality. It relies on standard features of model checkers, and could therefore easily be introduced in any such tool. For example, we are currently in the process of implementing it in JPF for the analysis of Java code [32] and in FLAVERS for the analysis of Ada code [14] and Java code [29]. The architecture of our framework is modular, so its components can easily be substituted by more efficient ones. To evaluate how useful our approach is in practice, we are planning its extensive application to real systems. However, our early experiments provide strong evidence in favor of this line of research.

In the future, we plan to investigate a number of topics including whether the learning algorithm can be made more efficient in our context; whether different algorithms would be more appropriate for generating the assumptions; whether we could benefit by querying a component and its environment at the same time or by implementing more powerful compositional rules. An interesting challenge will also be to extend the types of properties that our framework can handle to include liveness, fairness, and timed properties.

References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Compositionality: The Significant Difference - An International Symposium*, pages 23–60, Sept. 1997.
2. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proceedings of the Tenth International Conference on Computer-Aided Verification*, pages 521–525, June 28–July 2, 1998.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.
4. H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proceedings of the Second Workshop on Specification and Verification of Component-Based Systems*, Sept. 2003.
5. K. M. Chandy and J. Misra. *Parallel Program Design: a Foundation*. Addison-Wesley, 1988.
6. S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, Oct. 1996.
7. S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, Jan. 1999.
8. T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.
9. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 353–362, June 1989.
10. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning Assumptions for Compositional Verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

11. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Eighth European Software Engineering Conference held jointly with the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 109–120, Sept. 2001.
12. L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software*, pages 148–165, Oct. 2001.
13. T. Dean and M. S. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Aug. 1988.
14. M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.
15. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proceedings of the Eleventh European Symposium on Programming*, pages 262–277, Apr. 2002.
16. C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of the Tenth SPIN Workshop*, pages 213–224, May 2003.
17. D. Giannakopoulou, J. Kramer, and S. C. Cheung. Behaviour analysis of distributed systems using the Tracta approach. *Automated Software Engineering*, 6(1):7–35, July 1999.
18. D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the Seventeenth IEEE International Conference on Automated Software Engineering*, Sept. 2002.
19. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370, Apr. 2002.
20. O. Grumberg and D. E. Long. Model checking and modular verification. In *Proceedings of the Second International Conference on Concurrency Theory*, pages 250–265, Aug. 1991.
21. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of the Fifteenth International Conference on Computer-Aided Verification*, pages 262–274, July 2003.
22. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proceedings of the Tenth International Conference on Computer-Aided Verification*, pages 440–451, June 28–July 2, 1998.
23. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
24. R. D. Jeffords and C. L. Heitmeyer. A strategy for efficiently verifying requirements. In *Proceedings of the Ninth European Software Engineering Conference held jointly with the Eleventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 28–37, Sept. 2003.
25. C. B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–

332. IFIP: North Holland, 1983.

26. J.-P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 239–258, Apr. 1997.
27. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
28. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
29. G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, May 1999.
30. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, New York, 1984. Springer-Verlag.
31. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, Apr. 1993.
32. W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, pages 3–12, Sept. 2000.
33. Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.